Research Paper | Computer Science | E-ISSN No : 2454-9916 | Volume : 2 | Issue : 4 | April 2016

ERROR DETECTION IN BIG DATA ON CLOUD

Shweta Dolhare [1] | Snehal Gaikwad [1] | Poonam Ghorpade [1]

[1] Computer Engineering , BVCOEW, Pune, India- 411043.

ABSTRACT

Big sensor data is prevalent in both industry and scientific research applications where the data is generated with high volume and velocity it is difficult to process using on-hand database management tools or traditional data processing applications. Cloud computing provides a promising platform to support the addressing of this challenge as it provides a flexible stack of massive computing, storage, and software services in a scalable manner at low cost. But uploading the data in cloud can with high volume and velocity can cause damage to that file. So detecting such files from the data base is important. So in this project we are detecting the files with data tamper or error. We also propose to repair the tampered file to its original state and restore it again. Hence the Data in the cloud will remain unharmed.

KEYWORDS: Cloud Computing, Service Composition, Online Web services, Hadoop, MySQL, MapReduce.

Introduction:
We need to develop such a approach that will efficiently reduce time for detecting errors in big sensor data on cloud. If any error is found then it also involves error recovery & storing the data in original format. According to the error type and features from scale-free network we have proposed a time-efficient strategy for detecting and locating errors in big data sets on cloud . The main aim is to reduce the time required to detect the errors and to provide a error free transmission of data .

Materials and Methods:
Hardware Requirement:
1. 500GB HD
2. 8GB RAM

Software Requirement:
1. JAVA
2. MySQL
3. Hadoop

METHODS :
1.The Basic Idea Behind CRC Algorithms:
Where might we go in our search for a more complex function than summing? All sorts of schemes spring to mind. We could construct tables using the digits of pi, or hash each incoming byte with all the bytes in the register. We could even keep a large telephone book on-line, and use each incoming byte combined with the register bytes to index a new phone number which would be the next register value. The possibilities are limitless. However, we do not need to go so far; the next arithmetic step suffices. While addition is clearly not strong enough to form an effective checksum, it turns out that division is, so long as the divisor is about as wide as the checksum register. The basic idea of CRC algorithms is simply to treat the message as an enormous binary number, to divide it by another fixed binary number, and to make the remainder from this division the checksum. Upon receipt of the message, the receiver can perform the same division and compare the remainder with the "checksum" (transmitted remainder).

Example: Suppose the the message consisted of the two bytes (6,23) as in the previous example. These can be considered to be the hexadecimal number 0617 which can be considered to be the binary number 0000-0110-0001-0111. Suppose that we use a checksum register one-byte wide and use a constant divisor of 1001, then the checksum is the remainder after 0000-0110-0001-0111 is divided by 1001. While in this case, this calculation could obviously be performed using common garden variety 32-bit registers, in the general case this is messy. So instead, we'll do the division using good-'ol long division which you learnt in school (remember?). Except this time, it's in binary:

```
...0000010101101 = 00AD =  173 = QUOTIENT
      ____-___-___-___-
9= 1001 ) 0000011000010111 = 0617 = 1559 = DIVIDEND
DIVISOR     0000.,,....,.,,,
            ----.,,....,.,,,
            0000,,....,.,,,
            0000,,....,.,,,
            ----,,....,.,,,
            0001,....,.,,,
            0000,....,.,,,
            ----,....,.,,,
            0011....,.,,,
            0000....,.,,,
            ----....,.,,,
            0110...,.,,,
            0000...,.,,,
            ----...,.,,,
            1100..,.,,,
            1001..,.,,,
            ====..,.,,,
            0110.,.,,,
            0000.,.,,,
            ----.,.,,,
            1100,.,,,
            1001,.,,,
            ====,.,,,
            0111.,,,
            0000.,,,
            ----.,,,
            1110,,,
            1001,,,
            ====,,,
            1011,,
            1001,,
            ====,,
            0101,
            0000,
            ----
            1011
            1001
            ====
            0010 = 02 = 2 = REMAINDER
```

In decimal this is "1559 divided by 9 is 173 with a remainder of 2". Although the effect of each bit of the input message on the quotient is not all that significant, the 4-bit remainder gets kicked about quite a lot during the calculation, and if more bytes were added to the message (dividend) it's value could change radically again very quickly. This is why division works where addition doesn't. In case you're wondering, using this 4-bit checksum the transmitted message would look like this (in hexadecimal): 06172 (where the 0617 is the message and the 2 is the checksum). The receiver would divide 0617 by 9 and see whether the remainder was 2.

2. HUMMING CODE :
The key to the Hamming Code is the use of extra parity bits to allow the identification of a single error. Create the code word as follows:

1. Mark all bit positions that are powers of two as parity bits. (positions 1, 2, 4, 8, 16, 32, 64, etc.)

2. All other bit positions are for the data to be encoded. (positions 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, etc.)

3. Each parity bit calculates the parity for some of the bits in the code word. The position of the parity bit determines the sequence of bits that it alternately

checks and skips.

Position 1: check 1 bit, skip 1 bit, check 1 bit, skip 1 bit, etc. (1,3,5,7,9,11,13,15,...)

Position 2: check 2 bits, skip 2 bits, check 2 bits, skip 2 bits, etc. (2,3,6,7,10,11,14,15,...)

Position 4: check 4 bits, skip 4 bits, check 4 bits, skip 4 bits, etc. (4,5,6,7,12,13,14,15,20,21,22,23,...)

Position 8: check 8 bits, skip 8 bits, check 8 bits, skip 8 bits, etc. (8-15,24-31,40-47,...)

Position 16: check 16 bits, skip 16 bits, check 16 bits, skip 16 bits, etc. (16-31,48-63,80-95,...)

Position 32: check 32 bits, skip 32 bits, check 32 bits, skip 32 bits, etc. (32-63,96-127,160-191,...) etc.

4. Set a parity bit to 1 if the total number of ones in the positions it checks is odd. Set a parity bit to 0 if the total number of ones in the positions it checks is even.

Here is an example:
A byte of data: 10011010

Create the data word, leaving spaces for the parity bits: _ _ 1 _ 0 0 1 _ 1 0 1 0

Calculate the parity for each parity bit (a ? represents the bit position being set):

- Position 1 checks bits 1,3,5,7,9,11: ? _ 1 _ 0 0 1 _ 1 0 1 0. Even parity so set position 1 to a 0: 0 _ 1 _ 0 0 1 _ 1 0 1 0
- Position 2 checks bits 2,3,6,7,10,11: 0 ? 1 _ 0 0 1 _ 1 0 1 0. Odd parity so set position 2 to a 1: 0 1 1 _ 0 0 1 _ 1 0 1 0
- Position 4 checks bits 4,5,6,7,12: 0 1 1 ? 0 0 1 _ 1 0 1 0. Odd parity so set position 4 to a 1: 0 1 1 1 0 0 1 _ 1 0 1 0
- Position 8 checks bits 8,9,10,11,12: 0 1 1 1 0 0 1 ? 1 0 1 0. Even parity so set position 8 to a 0: 0 1 1 1 0 0 1 0 1 0 1 0
- Code word: 011100101010.

3. SECURE HASH ALGORITHM:
The **Secure Hash Algorithm** is a family of cryptographic hash functions published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS), including:

- **SHA-0:** A retronym applied to the original version of the 160-bit hash function published in 1993 under the name "SHA". It was withdrawn shortly after publication due to an undisclosed "significant flaw" and replaced by the slightly revised version SHA-1.
- **SHA-1:** A 160-bit (20 byte)hash function which resembles the earlier MD5 algorithm. This was designed by the National Security Agency (NSA) to be part of the Digital Signature Algorithm.
- **SHA-2:** A family of two similar hash functions, with different block sizes, known as SHA-256 and SHA-512. They differ in the word size; SHA-256 uses 32-bit words where SHA-512 uses 64-bit words. There are also truncated versions of each standard, known as SHA-224, SHA-384, SHA-512/224 and SHA-512/256. These were also designed by the NSA.
- **SHA-3** A hash function formerly called Keccak, chosen in 2012 after a public competition among non-NSA designers. It supports the same hash lengths as SHA-2, and its internal structure differs significantly from the rest of the SHA family.

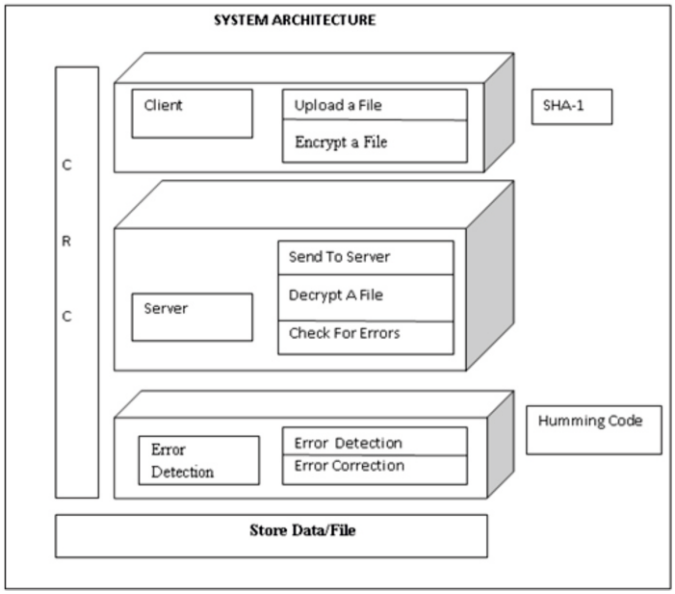
Results and Discussion:
In this paper, our work described how errors can detected and corrected and data retransmit to the storage criteria cloud. First, it aims at transmission of data in encrypted format with the help of SHA-1 algorithm from sender to receiver. Second, to improve the scalability and efficiency in "Big Data" environment, we have implemented it on a MapReduce framework in Hadoop platform. Finally, the data appears at receiver side get decrypted by the same algorithm. Then it will check for the errors, if any error found then it will retransmit that respective data. It will only detects specific types of errors.

Acknowledgments:
First and foremost we would like to express our gratitude to Prof. S. B. Jadhav, Our internal guide, for her guidance and support throughout the project. Without her cooperation, it would have been extremely difficult for us to complete the project part of this semester.

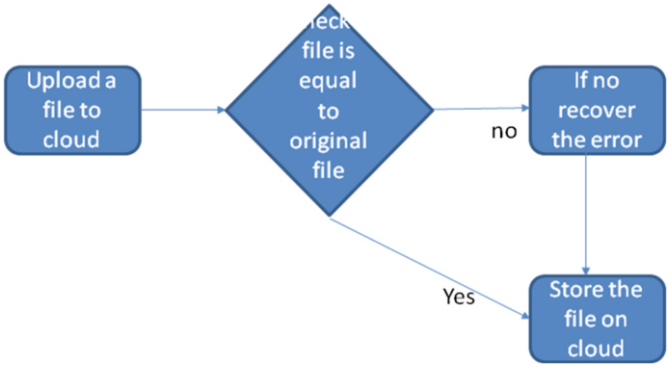
We would also like to thank the HOD of the computer department, Prof. D. D. Pukale, as well as the entire teaching and non-teaching staff of the Computer Department for giving us an opportunity to work on such an exciting project.

Last but not the least; we are extremely grateful to our family, friends and colleagues who have supported us right from the inception of the project. Thanks for all your encouragement and support.

Figures:
System Architecture :



Block Diagram :



REFERENCES

1. S. Tsuchiya, Y. Sakamoto, Y. Tsuchimoto, and V. Lee, "Big Data Processing in Cloud Environments," FUJITSU Science and Technology J., vol. 48, no. 2, pp. 159-168, 2012.
2. S. Sakr, A. Liu, D. Batista, and M. Alomari, "A Survey of Large Scale Data Management Approaches in Cloud Environments," IEEE Comm. Surveys & Tutorials, vol. 13, no. 3, pp. 311-336, Third Quarter 2011.
3. M.C. Vuranand and I.F. Akyildiz, "Error Control in Wireless Sensor Networks: A Cross Layer Analysis," IEEE Trans. Networking, vol. 17, no. 4, pp. 1186-1199, Aug. 2009.
4. C. Liu, J. Chen, T. Yang, X. Zhang, C. Yang, R. Ranjan, and K. Kotagiri, "Authorized public auditing of dynamic big data storage on cloud with efficient verifiable fine-grained updates," IEEE Trans. Parallel and Distributed Systems, vol. 25, no. 9, pp. 2234–2244, Sept. 2014
5. "Sensor Cloud," http://www.sensorcloud.com/, accessed on 30, Aug. 2013.
6. M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski,G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A View of Cloud Computing," Comm. the ACM, vol. 53, no. 4, P p. 50-58, 2010.
7. "Big Data: Science in the Petabyte Era: Community Cleverness Required," Nature, vol. 455, no. 7209, p. 1, 2008.